# Programming in Haskell – Homework Assignment 5

UNIZG FER, 2014/2015

Handed out: November 2, 2014. Due: November 9, 2014 at 23:59

*Note:* Define each function with the exact name and the type specified. You can (and in most cases you should) define each function using a number of simpler functions. Provide a type signature above each function definition and comment the function above the type signature. Unless said otherwise, a function may not cause runtime errors and must be defined for all of its input values. Use the `error` function for cases in which a function should terminate with an error message. Problems marked with a star ($\star$) are optional.

Each problem is worth a certain number of points. The points are given at the beginning of each problem or subproblem (if they are scored independently). These points are scaled, together with a score for the in-class exercises, if any, to 10. Problems marked with a star ($\star$) are scored on top of the mandatory problems, before scaling. The score is capped at 10, but this allows for a perfect score even with problems remaining unsolved.

1. *(1 pt)* Define your own version of `Data.List.Intercalate`, called `intercalate'`. It must be implemented using explicit recursion.

   ```
   intercalate' :: [a] -> [[a]] -> [a]
   intercalate' xs yss == intercalate xs yss
   ```

2. *(1 pt)* Define the following functions that work with discrete random variables. You can assume the probabilities will always be valid ($0 \leq p_i \leq 1$, $\forall i$ and $\sum_{i=1}^{N} p_i = 1$). The random variables are defined as follows:

   ```
   type Probability = Double
   type DiscreteRandVar = [(Int, Probability)]
   x :: DiscreteRandVar
   x = [(1, 0.2), (2, 0.4), (3, 0.1), (4, 0.2), (5, 0.05), (6, 0.05)]
   ```

   (a) Define an explicitly recursive function `mean` and an accumulator-style recursive function `mean'` that calculate the mean (or expected value) of a discrete random variable, defined as $\sum_{i=1}^{N} x_i \cdot p_i$.

   ```
   mean  :: DiscreteRandVar -> Double
   mean' :: DiscreteRandVar -> Double
   mean x ⇒ 2.65
   mean' x ⇒ 2.65
   ```

   (b) Define an explicitly recursive function `variance` and an accumulator-style recursive function `variance'` that calculate the variance of a discrete random variable. Given the mean of the variable as $\mu_x$, it is defined as $\sum_{i=1}^{N}(x_i - \mu_x)^2 \cdot p_i$.

   ```
   variance  :: DiscreteRandVar -> Double
   variance' :: DiscreteRandVar -> Double
   variance x ⇒ 1.9275
   variance' x ⇒ 1.9275
   ```

(c) Define an explicitly recursive function `probabilityFilter` and an accumulator-style recursive function `probabilityFilter'` that take a probability and a random variable and return a list of values that have at least the given probability of appearing, *in the same order* in which they appear in the random variable definition.

```
probabilityFilter  :: Probability -> DiscreteRandVar -> [Int]
probabilityFilter' :: Probability -> DiscreteRandVar -> [Int]
probabilityFilter 0 x == x
probabilityFilter' 0 x == x
probabilityFilter 0.2 x ⇒ [1,2,4]
probabilityFilter' 0.2 x ⇒ [1,2,4]
```

3. *(1 pt)* Often we need to split up a list into smaller chunks. Define three functions to do just that.

(a) Define a function `chunk` that splits up a list `xs` into sublist of length `n`. If the length of `xs` is not a multiple of `n`, the last sublist will be shorter than `n`. Define the function using explicit recursion.

```
chunk :: Int -> [a] -> [[a]]
chunk 2 "shadowless" ⇒ ["sh","ad","ow","le","ss"]
chunk 4 "shadowless" ⇒ ["shad","owle","ss"]
chunk 11 "shadowless" ⇒ ["shadowless"]
chunk 0 "shadowless" ⇒ []
```

(b) Define a function `chunkBy` that splits up a list `xs` into sublists of lengths given in a list of indices `is`. If the lengths in `is` do not add up to the length of `xs`, the remaining part of `xs` will remain unchunked. Define the function using explicit recursion.

```
chunkBy :: [Int] -> [a] -> [[a]]
chunkBy [1,2,3] "shadowless" ⇒ ["s","ha","dow"]
chunkBy [11,2] "shadowless" ⇒ ["shadowless"]
chunkBy [3,0,3,4] "shadowless" ⇒ ["sha","dow","less"]
chunkBy [] "shadowless" ⇒ []
```

(c) Define a function `chuckInto` that splits up a list `xs` into `n` sublists of equal length. If the length of `xs` is not divisible by `n`, chunk the remainder into the last sublist. Define the function using explicit recursion.

```
chuckInto :: Int -> [a] -> [[a]]
chunkInto 5 "shadowless" ⇒ ["sh","ad","ow","le","ss"]
chunkInto 3 "shadowless" ⇒ ["sha","dow","less"]
chunkInto 1 "shadowless" ⇒ ["shadowless"]
chunkInto 0 "shadowless" ⇒ []
chunkInto 11 "shadowless"
⇒ ["s","h","a","d","o","w","l","e","s","s"]
```

4. *(1 pt)* Define a function `rpnCalc` that takes a mathematical expression written in Reverse Polish notation and calculates its result. The expression is limited to 1–digit positive integers and operators `+`,`-`,`*`,`/`, and `^` where `/` is integer division and `^` is exponentiation.

```
rpnCalc :: String -> Int
rpnCalc "32-1+" ⇒ 2
```

```
rpnCalc "321-+5-" ⇒ -1
rpnCalc "42^2/" ⇒ 8
rpnCalc "35*2/" ⇒ 7
rpnCalc "35*7-3^43*/" ⇒ 42
rpnCalc "" ⇒ 0
rpnCalc "33++" ⇒ error "Invalid RPN expression"
```

5. *(1 pt)*

   (a) Define a function `gcd'` that calculates the greatest common divisor of two integers, using explicit recursion and the Euclidean algorithm.

   ```
   gcd' :: Int -> Int -> Int
   gcd' 75 125 ⇒ 25
   gcd' 1592 368 ⇒ 8
   gcd' 17 4 ⇒ 1
   gcd' (-153) 24 ⇒ 3
   gcd' (-2541) (-1134) ⇒ 21
   gcd' 15 0 ⇒ 15
   ```

   (b) Define a function `gcdAll` that calculates the greatest common divisor of an arbitrary number of integers given in a list.

   ```
   gcdAll :: [Int] -> Int
   gcdAll [3,15] ⇒ 3
   gcdAll [21,49,14,(-14)] ⇒ 7
   gcdAll [52] ⇒ 52
   gcdAll [7060, 16944, 19768] ⇒ 1412
   gcdAll [] ⇒ error "Cannot compute gcd of an empty list"
   ```

   (c) Define a function `extendedGcd` which uses the extended Euclidean algorithm to calculate the Bezout coefficients along with the gcd. Given the constants `a` and `b`, it calculates `x`, `y` and `gcd(a,b)` that satisfy the expression `a*x + b*y = gcd(a,b)`, returning them in a tuple, in that order.

   ```
   extendedGcd :: Int -> Int -> (Int,Int,Int)
   extendedGcd 240 46 ⇒ (-9,47,2)
   extendedGcd 314 159 ⇒ (-40,79,1)
   extendedGcd 1245 (-1603) ⇒ (300,233,1)
   ```

6. *(1 pt)* Implement a function `isBipartite` that takes an unweighted graph represented as an adjacency list and checks whether the given graph is a bipartite graph. A graph is considered bipartite if its vertices can be split into two disjoint sets such that there is no edge which connects two vertices from the same set.

   ```
   type AdjacencyList = [Int]
   type Graph = [AdjacencyList]
   ```

   ```
   isBipartite :: Graph -> Bool
   isBipartite [[2],[1]] ⇒ True
   isBipartite [[4,5,6],[4,5,6],[4,5,6],[1,2,3],[1,2,3],[1,2,3]] ⇒ True
   isBipartite [[2,3],[3,1],[1,2]] ⇒ False
   ```

7. *(1.5 pts)* Define a function `permutations'` that, given a list, returns a list of all its permutations. Implement it using explicit recursion. The ordering of the list of permutations is irrelevant!

```
permutations' :: [a] -> [[a]]
permutations' "hi" ⇒ ["hi", "ih"]
permutations' [1,2,3]
⇒ [[1,2,3],[2,1,3],[3,2,1],[2,3,1],[3,1,2],[1,3,2]]
permutations' [] ⇒ [[]]
```

8.⋆ *(2 pts)* Imagine a group of differently–sized frogs living in a swamp with only three lily pads numbered from 1 to 3. Every morning, frogs meet at the first lily pad and spend all day trying to reach the third one, thereby abiding the following rules:

- A frog cannot jump between the first and the third lily pad as they are too far apart;
- A frog can jump from a lily pad only if it is the smallest frog on that lily pad;
- A frog can jump on a lily pad only if it is smaller than all frogs on that lily pad.

Your job is to define a function `frogJumps` that, given a number of frogs `n`, computes the minimal number of jumps necessary for all `n` frogs to reach the third lily pad.

```
frogJumps :: Int -> Integer
frogJumps 1 ⇒ 2
frogJumps 2 ⇒ 8
frogJumps 5 ⇒ 242
frogJumps 20 ⇒ 3486784400
```